



CE Linux Forum

CELF検討技術の適用事例

2006年6月30日

WILLCOMコアモジュールフォーラム

組込Linux WG

近藤 (jyunji.kondo@wcmf.jp)



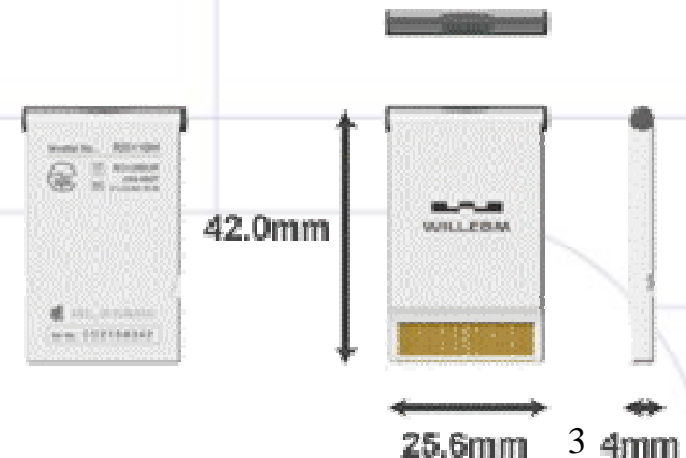
Agenda

- CELF検討技術
- SGWPへの技術適用
 - Bootchart
 - Kernel Function Trace
 - User Function Trace



WCMFとは

- WILLCOMコアモジュールフォーラム
- 株式会社ウィルコムが開発し提案するW-SIMを核として、各パートナーが発展できる非営利団体
- 詳細は <http://www.wcmf.jp/> にて



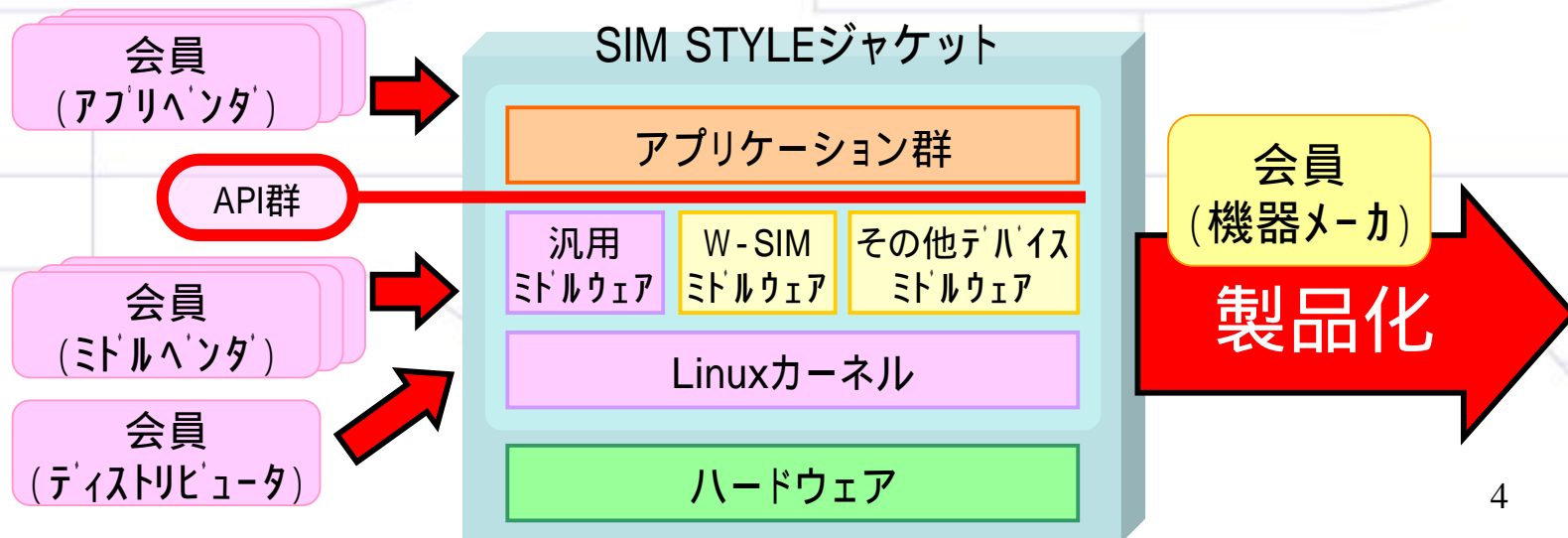


WCMF 組込Linux WGとは

- 目的

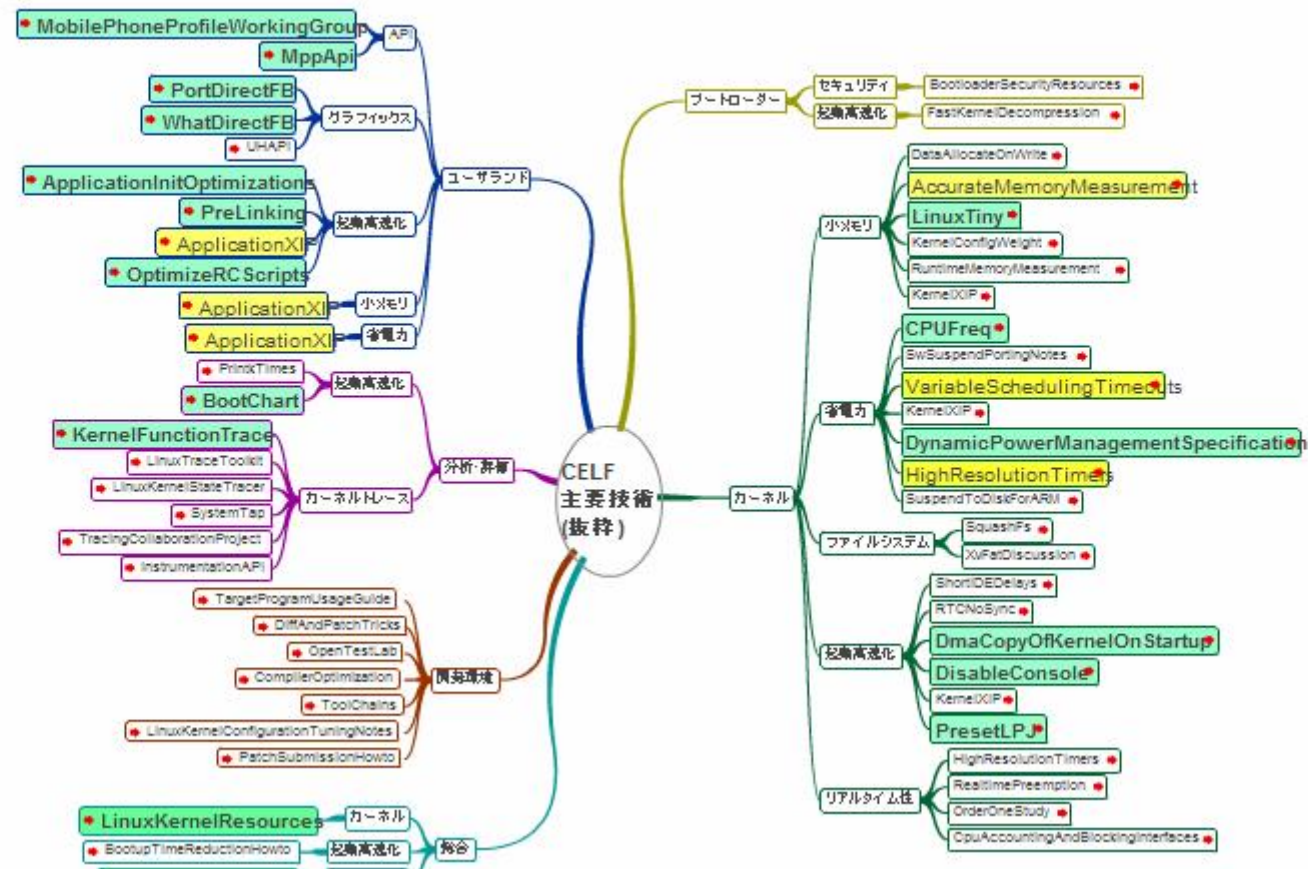
- Linux OSを採用したSIM STYLEジャケットの研究
- アプリケーションの普及促進

詳細は<http://www.wcmf.jp/soshiki/01.html> にて





CELF検討技術





SGWPとは



Sandgate WP
Sandgate W-Sim Phone

- プロセッサ:インテルPXA270(416MHz)
- OS:Linux(VER.2.6.15)
- メモリ:SDRAM 64MB, Flash ROM 128MB
- カメラ:1.3MピクセルのCCDカメラ
- LCD:2.2インチTFT液晶QVGA(26万色)
- W-SIMスロット
- 外部インタフェース:
USB1.1(Client)MiniBコネクタ, MiniSDスロット
- 赤外線通信:IrDA
- Bluetooth
- 4-wayナビゲーションボタン搭載
- キー照明付き

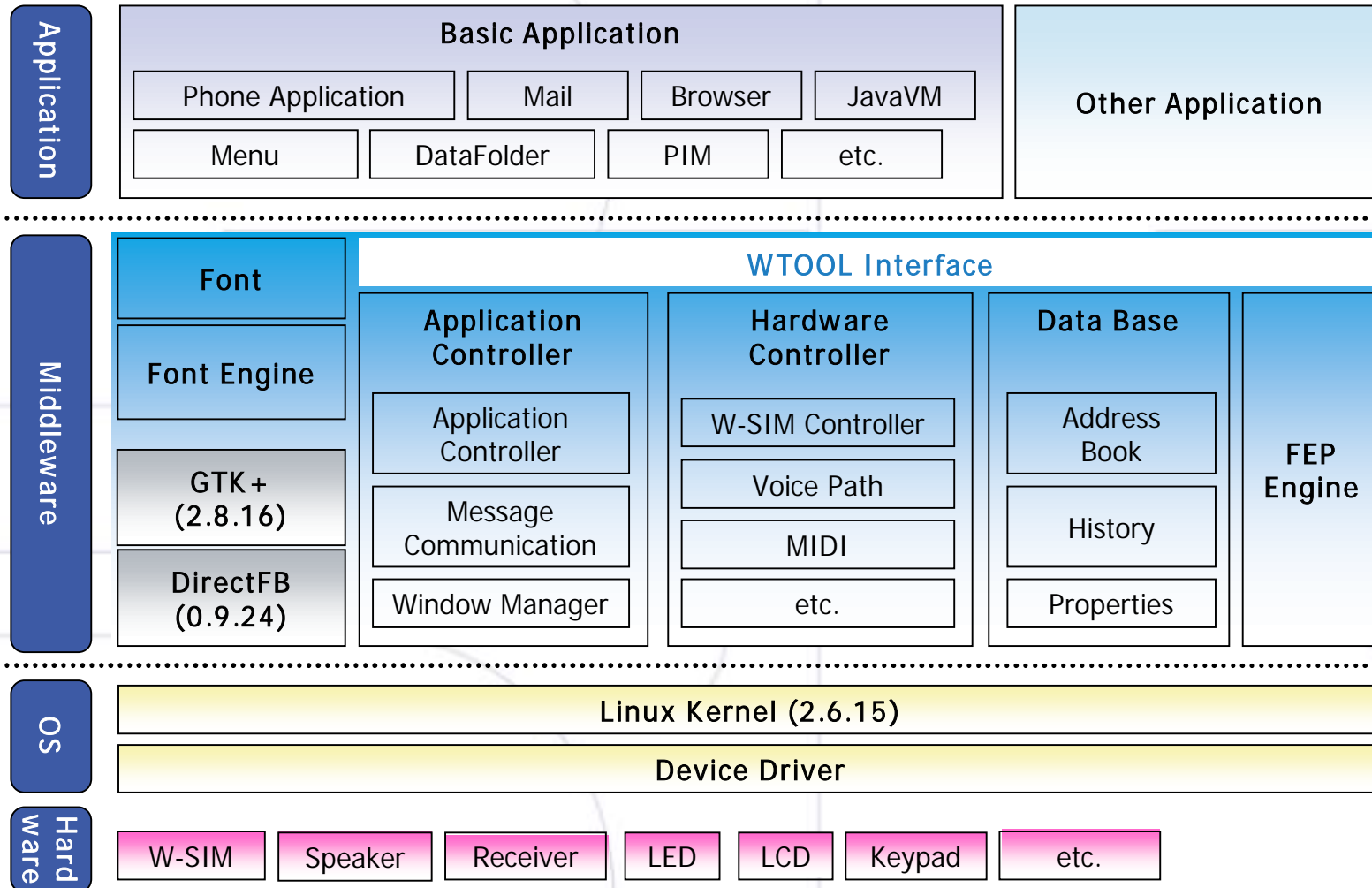
- 着信音用マイクロスピーカ
- 通話用レシーバ
- 通話用マイク
- ハンズフリー:平型ジャック搭載
- 着信LED:約32000色(グラデーション制御可能)
- 着信通知用バイブレータ

- リチウムバッテリー:3.7V 1300mAh
- 充電:USBポートから給電
- ACアダプタ:USBポートに接続
- 強制リセットスイッチ
- デバッグ用ボード(JTAG, シリアル, LAN)
- ハードウェアを追加するための拡張ポート搭載





SGWPのソフトウェアスタック





開発時の課題

- 起動が遅い
- キー押下からアプリの起動までに時間がかかる

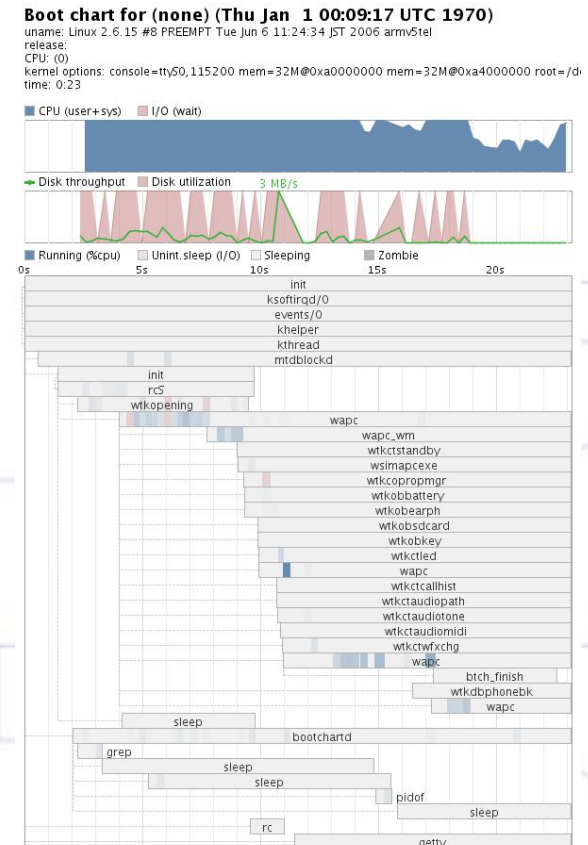
現状分析が必要！

- ツール
 - Bootchart
 - KernelFunctionTrace



Bootchart

- CELF Wikiでも紹介あり(*1)
- システム起動中に一定間隔(*2)で以下の情報を収集
 - プロセスの状態 (R,D,S,Z)
 - Disk I/Oの統計情報
 - プロセス課金情報
- 収集した情報を、PC上でグラフ化



* 1 <http://tree.celinuxforum.org/pubwiki/moin.cgi/BootChart>

* 2 オリジナルでは、0.2秒間隔



Bootchart

名前	psコマンドのマニュアルより	色	/proc/[PID]/stat内でのプロセスの状態コード
Running (%cpu)	実行中または実行可能状態 (実行キューにある)	#ffcb00 + 128 + %CPU * 128 色が濃いほどCPUを使用している	R
Unint.sleep	割り込み不可能なスリープ状態 (通常 IO 中)	gray	D
Sleeping	割り込み可能なスリープ状態 (イベントの完了を待っている)	light gray	S
Zombie	終了したが、親プロセスによって 回収されなかった、消滅した (ゾンビ) プロセス	dark gray	Z



Kernel Function Trace

- メカニズム
 - -finstrument-functionsオプションを付けてカーネルをコンパイルする
 - 各カーネル関数の出入口で、プロファイル用の関数を呼び出すコードが自動挿入される
 - プロファイル用の関数で、以下を記録する
 - 関数が呼び出された時間
 - 復帰するまでに費やした時間
 - 記録したデータは、procファイルシステムから読み取ることができる
 - 採取したデータは、専用ツールで加工して分析する

参考URL: <http://tree.celinuxforum.org/pubwiki/moin.cgi/KernelFunctionTrace>

```
Trace
-----
start_kemel
| printk
| | vprintk
| printk
| | vprintk
| setup_arch
| | setup_processor
| | | printk
| | | | vprintk
| | | | vsprintf
| | | | vsprintf
| | setup_machine
| | | printk
| | | | vprintk
| | parse_tags
| | | parse_tag
| | | | parse_tag_endl ine
| | | parse_endl ine
| | paging_init
| | | build_mem_type_table
| | | | printk
| | | | | vprintk
| | | bootmem_init
| | | | bootmem_init_node
| | | | | init_bootmem_node
| | | | | | init_bootmem_core
| | | | | | free_bootmem_node
| | | | | | free_bootmem_core
| | | | | | free_bootmem_node
| | | | | | free_bootmem_core
| | | | | | reserve_node_zero
| | | | | | reserve_bootmem_node
| | | | | | | reserve_bootmem_core
| | | | | | | free_area_init_node
```



Kernel Function Trace – 詳細

- 計時方法
 - Xscaleプロセッサのパフォーマンスカウンタ(CCNT)を利用
 - カーネル圧縮イメージを展開する前に、ゼロクリアして計時開始

```
--- linux-2.6.17/arch/arm/boot/compressed/head.S 2006-06-18 10:49:35.000000000 +0900
+++ linux-2.6.17-kft/arch/arm/boot/compressed/head.S 2006-06-21 05:53:29.000000000 +0900
@@ -118,6 +118,10 @@
        .word      _edata                                @ zImage end address
1:      mov        r7, r1                                @ save architecture ID
        mov        r8, r2                                @ save atags pointer
+#if defined(CONFIG_KFT) && defined(CONFIG_ARCH_PXA)
+      mov        r0, #0x0d
+      mcr p14, 0, r0, c0, c1, 0        @ reset clock counter to '0x0' and CCNT
+                                       @ counts every 64th processor clock cycle
+#endif /* CONFIG_KFT, CONFIG_ARCH_PXA */

#ifdef __ARM_ARCH_2__
/*
```



Kernel Function Trace – 詳細

- 計時方法(つづき)
 - プロファイル用の関数から呼び出される
kft_readclock()関数は、CCNTの値を返す

```
--- linux-2.6.17/kernel/kft.c 1970-01-01 09:00:00.000000000 +0900
+++ linux-2.6.17-kft/kernel/kft.c 2006-06-21 05:53:29.000000000 +0900
:
+#ifdef CONFIG_ARCH_PXA
+static inline u32 read_counter(void)
+{
+    u32 val = 0;
+
+    // CCNT:
+    __asm__ __volatile__ ("mrc p14, 0, %0, c1, c1, 0" : "=r" (val));
+
+    return val;
+}
+
+static inline unsigned long __noinstrument kft_readclock(void)
+{
+    return read_counter();
+}
+#endif // CONFIG_ARCH_PXA
```



Kernel Function Trace – 詳細

- データ加工の流れ
 - ターゲット側にて
 1. `cat /proc/kft_data > kft_data.txt`
 - PC側にデータを転送して
 2. `scripts/addr2sym < kft_data.txt ¥
-m system.map > kft_data-sym.txt`
 3. `scripts/kd -s 1 kft_data-sym.txt ¥
> kft_data-sym-s1.txt`
(ローカル時間でソート)
 4. `scripts/kd -c kft_data-sym.txt ¥
> kft_data-sym-c.txt`
(関数呼出のツリー生成)



ここでさらに課題

- 時間がかかるのはカーネルばかりではない！
- ユーザ空間での処理も分析が必要
- KernelFunctionTraceのようにできないものか
- そこで...



User Function Trace (仮称)

- 方針
 - Kernel Function Traceの考えを、ユーザ空間にも適用する
 - アプリ、ライブラリを-finstrument-functionsオプションを付けてコンパイルする
 - 同じようなものは再発明したくない
 - Kernel Function Trace用の分析ツールを流用したい



User Function Trace (つづき)

- しかし、色々課題が...
- 1. プロファイル用の関数で、どのように記録するか？
 - カーネルは1つ
 - 1つの構造体配列に順番に
 - プロセスは複数
 - 1つの構造体配列に同期を取りながら？ (面倒くさそう)
 - プロセス毎に複数の構造体配列に？ (どう結合する？)



User Function Trace (つづき)

- 課題 (つづき)

2. 記録するための構造体配列の大きさは？

- KFTは動的に指定可能

```
# echo "new filter mintime 10 logentries 200000 ¥  
trigger start entry c0029bb4 end" > /proc/kft  
# echo "start" > /proc/kft
```

- アプリで静的に配列を確保すると...

- アプリのメモリ使用量が増えてOOM Killerの危機が



User Function Trace (つづき)

- 課題 (つづき)
- 3. 既存のアプリ、ライブラリ自身のコードに手を入れたくない



User Function Trace (つづき)

- 課題 (つづき)
- 4. ダイナミックリンク・ライブラリをどう扱うか？
 - ダイナミックリンク・ライブラリ自身のシンボル情報は
相対アドレス
 - 実際のシンボルアドレスはアプリを動かすまで分からない



User Function Trace (つづき)

- 課題 (つづき)

5. システムコール呼出、ディスパッチ時の時間調整

- アプリ内の関数からシステムコールを発行した場合
 - 関数内の処理時間にカーネル処理時間が紛れ込む
- アプリがタイムスライスを使い果たしてディスパッチされた場合
 - 関数内の処理時間に他のアプリの処理時間が紛れ込む



User Function Trace (つづき)

- 課題まとめ
 1. プロファイル用の関数で、どのように記録するか？
 2. 記録するための構造体配列の大きさは？
 3. 既存のアプリ、ライブラリ自身のコードに手を入れたくない
 4. ダイナミックリンク・ライブラリをどう扱うか？
 5. システムコール呼出、ディスパッチ時の時間調整



User Function Trace (つづき)

- 課題 1、2 の解決策
 1. 記録用配列は持たない。
プロファイル関数で採取した情報は、プロセス毎にファイルを分けて書き出す。
 - あらたな問題
 - KFTは、固定配列を前提に関数の処理時間を算出して記録する
 1. 関数入り口で開始時間を配列に記録
 2. 出口で、入り口の時間を探す。処理時間を算出して配列に記録する
 - 同様の処理ができない
 - 解決策
 - 書き出したファイルから、後でKFT形式のデータに変換する



User Function Trace (つづき)

- 課題3の解決策
 1. コンパイルオプション追加とデバッグ用のモジュールを追加リンクする。
ソースコードは修正しない。
 - -finstrument-functionsを追加する
 - instrument-functions.oを作成して追加リンクする



User Function Trace (つづき)

- 課題3の解決策(つづき)
- instrument-functions.oの中身
 - gccのコンストラクター関数で、情報を書き出すファイルを開く
 - gccのデコンストラクター関数で、ファイルをクローズする
- 詳しくは、ソースコードで説明

[参考] http://www-06.ibm.com/jp/developerworks/linux/050722/j_1-graphvis.html



User Function Trace (つづき)

- 課題4の解決策
 1. アプリ実行時にプロセスのメモリマップを保存する
 2. ダイナミックリンク・ライブラリのマッピングアドレスと、ライブラリ自身のシンボル情報から、実行時のシンボルアドレスを算出する
- 詳しくはソースコードで説明



User Function Trace (つづき)

- データ加工の流れ
 - ターゲット側にて
 1. アプリが書き出したファイルをPC側に転送する
 - “コマンド名.プロセスID”
 - プロファイル関数を書き出した関数出入口のデータ
 - “コマンド名.プロセスID.maps”
 - プロセスのメモリマップ。共有ライブラリのリンク位置情報の取得用



User Function Trace (つづき)

- データ加工の流れ(つづき)

- PC側にて

```
2.for i in *.maps
do
buildmap $i ¥
cvs/cmd-lib/sys-
root/usr/lib:/opt/gnu/xscale/i686-pc-linux-
gnu/arm-xscale-linux-gnu/sys-
root/lib:/opt/gnu/xscale/i686-pc-linux-
gnu/arm-xscale-linux-gnu/sys-root/usr/lib ¥
> $(basename $i .maps).map
done
```

System.map相当のシンボル情報
ファイルを生成
("コマンド名.プロセスID.map")

```
3.for i in $(ls | egrep '[0-9]+$'); do mktracelog $i; done
```

```
4.mktotallog
```



User Function Trace (つづき)

- 今後の課題
 - 課題5の解決
 - システムコール呼出、ディスパッチ時の時間調整
 - もう少しリッチな可視化



Thank you!